CECS 342 - Lab assignment 2 - Parser

Due date: Monday, February 26

Team Members: Bryan Tineo & Maxwell Guillermo

Completion of Lab Assignment:

Both team members contributed equally and collaborated throughout the completion of the lab assignment.

Code lab2.c

```c
/* A lexical analyzer system for simple
arithmetic expressions */
#include <stdio.h>
#include <ctype.h>

/* Global declarations */
/* Variables */
int charClass;

//  if given this:  (sum + 47) / total
char lexeme[100]; // <-- [(,s,u,m,+,4,7,),/,t,o,t,a,l] This array is actually a whole
string. According to c/c++ this will be treated as both a whole null-terminated string
and a list of characters
char nextChar;
int lexLen; //<- initialize lexlen to keep trak of the length of the whole input from
txt
int token;
int nextToken;
FILE *in_fp, *fopen();

/* Function declarations */
void addChar();
void getChar();
void getNonBlank();
```

```
// update for lab2
// function declartions for expr, term, and factor
/* Parsing functions declarations */
void expr();
void term();
void factor();

int lex();

/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99

/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26

// update main function to call expr() function

/*****************************************************/
/* main driver */
int main()
{
     /* Open the input data file and process its contents */
     if ((in_fp = fopen("Test3.txt", "r")) == NULL)
           printf("ERROR - cannot open front.in \n");
     else
     {
           getChar();
           do
           {
                 lex();
                 // New implementation after lex
                 expr();
```

```
            } while (nextToken != EOF);
      }
      return 0;
}

/*****************************************************/
/* lookup - a function to lookup operators and parentheses
and return the token */
int lookup(char ch)
{
      switch (ch)
      {
      case '(':
            addChar();
            nextToken = LEFT_PAREN;
            break;
      case ')':
            addChar();
            nextToken = RIGHT_PAREN;
            break;
      case '*':
            addChar();
            nextToken = MULT_OP;
            break;
      case '/':
            addChar();
            nextToken = DIV_OP;
            break;
      case '+':
            addChar();
            nextToken = ADD_OP;
            break;
      case '-':
            addChar();
            nextToken = SUB_OP;
            break;
      case '=':
            addChar();
            nextToken = ASSIGN_OP;
            break;
      }
```

```
    return nextToken;
}

/*****************************************************/
/* addChar - a function to add nextChar to lexeme */
void addChar()
{
    if (lexLen <= 98)
    {
        // At the start of the program
        // lexLen = 0 that was assigned on lex function
        // when we are her
        // We are actually doing this:
        lexeme[lexLen++] = nextChar; // ex) when lexLen = 0  than, lexeme[0] = "("
        // After assigning "(" into the zero position the ++ (post i-increment) on
lexLen++ will update the Lexlen
        // After assinging to 0 position the "lexLen" will become, in this case,
were we will store the next character
        // However, since c/c++ are known for null terminated string. Than we need
to create a null value after this added character
        lexeme[lexLen] = 0; // -> lexeme[1] =0

        // So at the first itration the lexeme array would look like this
        // lexeme[100] = ["(",0]
        // c/c++ treats this array as an array of chracters
    }
    else
        printf("Error - lexeme is too long \n");
}

/*****************************************************/
/* getChar - a function to get the next character of
input and determine its character class */
void getChar()
{
    if ((nextChar = getc(in_fp)) != EOF)
    {
        if (isalpha(nextChar))
            charClass = LETTER;
        else if (isdigit(nextChar))
            charClass = DIGIT;
```

```
            else
                  charClass = UNKNOWN;
      }
      else
            charClass = EOF;
}

/*****************************************************/
/* getNonBlank - a function to call getChar until it
returns a non-whitespace character */
void getNonBlank()
{
      // ignoring white characters
      while (isspace(nextChar))
            getChar();
}
/*
******************************************************/
/* lex - a simple lexical analyzer for arithmetic
expressions */

int lex()
{
      lexLen = 0;     // length of lexeme
      getNonBlank(); //

      switch (charClass)
      {
      /* Parse identifiers */
      case LETTER:
            addChar();
            getChar();
            while (charClass == LETTER || charClass == DIGIT)
            {
                  addChar();
                  getChar();
            }
            nextToken = IDENT;
            break;
      case DIGIT:
```

```
            addChar();
            getChar();
            while (charClass == DIGIT)
            {
                  addChar();
                  getChar();
            }
            nextToken = INT_LIT;
            break;

      case UNKNOWN:
            lookup(nextChar);
            getChar();

            break;
      /* EOF */
      // handles the end-of-file character
      case EOF:
            nextToken = EOF; // <-- this assigns -1 as token for EOF
            lexeme[0] = 'E';
            lexeme[1] = 'O';
            lexeme[2] = 'F';
            lexeme[3] = 0; // Null-terminate the string
            break;
      } /* End of switch */
      printf("Next token is: %d, Next lexeme is %s\n", nextToken, lexeme);
      return nextToken;

} /* End of function lex */

void expr()
{
      printf("Enter <expr>\n");

      term(); // parse first term

      // <expr> -> <term> {(+ | -) <term>}
      while (nextToken == ADD_OP || nextToken == SUB_OP)
      {
            lex();  // we will get the next token
            term(); // parse that first token
      }
```

```
    // This function parses the first term and then, as long as the next token is
    // either a '+' or '-', it gets the next token and parses the next term.

    printf("Exit <expr>\n");
}

void term()
{
    printf("Enter <term>\n");

    factor(); // parse first factor

    // Parses strings in the language generated by the rule:
    // <term> -> <factor> {(* | /) <factor>}
    while (nextToken == MULT_OP || nextToken == DIV_OP)
    {
          lex();    // we will get the next token as an operator to indicate what i
snext
          factor(); // parse that first factor
    }

    // This function parses the first factor and then, as long as the next token is
    // either '*' or '/', it gets the next token and parses the next factor.

    printf("Exit <term>\n");
}

// Factor ends the definition or hwo would the non-terminal symbol would be
interpreted, it is the one creating the leaf of the parsing tree
void factor()
{
    printf("Enter <factor>\n");
    // Parses strings in the language generated by the rule:
    // <factor> -> id | int_constant | ( <expr> )
    // #define INT_LIT 10 #define IDENT 11
    /*
    <expr>
    |        \
    <term>     +
    |          \
    "10"       <term>
```

```
                |
                <factor>
                |     \
                "("    <expr>
                      /   |   \
                <term>   *   <term>
                |            |
                "5"          "4"
                |
                ")"
    */

    /*
    An EBNF description of simple arithmetic expressions:
    <expr>  -> <term> {(+ | -) <term>}
    <term>   -> <factor> {(* | /) <factor>}
    <factor> -> id | int constant | ( <expr> )
    */

    // in the diagram above, we need to check the nextToken in our expression
    // nexToken can be either an integer, identifier, or parenthesis
    // if the left branch of factor is either a terminal symbol(either variable or
intenger)
    // check for another expression or empty lexeme
    // or if the left branch has an open parenthesis this indicates a start of an new
expression
    // parse through the expression until closing parenthesis

    if (nextToken == IDENT || nextToken == INT_LIT)
    {
          // so if we know that is an id(variable) or and int(number) that means that
we reached the terminal symbol
          // Than we have to go to the continuation of what is on the right of the id
or int
          // We want to stop and go to the next part of the lexeme by using lex
          lex(); // we want to go to the lex until there is nothing to parse
    }
    // if the left branch of factor is neither a terminal symbol check for starting
parenthesis
    else if (nextToken == LEFT_PAREN)
    {
          // Continue getting more non-terminal symbols ex) <expr>, <term>,<factor>
```

```
            lex(); // goes to the next part of the lexeme
            // lex helps expr() in order for the expr() to identify what type of
non-terminal symbol is the next token
            expr(); // recursive call method for parsing the rest of the lexeme
expression(creating another root three inside this tree)
            // In this case, when this recursive function "expr()" ends.
            // Once this function returns, that mean that we are at the end of the
expression
            // meaning that we need to check for the closing parenthesis, that means
that the next token after the termination of this function should be a closing
parenthesis
            if (nextToken == RIGHT_PAREN)
            {
                  // We need to continue checking for more tokens inside the file
                  // this lex is necessary even if there is nothing after the
parenthesis since it will allow us to terminate the program
                  lex(); // this will also check whether we are already at the end of
the file in the next step of this current one
            }
            else
            {
                  printf("Error: Missing closing parenthesis in file\n");
            }
      }
      else
      {
            // than there might be an error on the actuall sytax
            // lets see...
            printf("Error: Syntax token\n");
      }

      // This function determines which right-hand side to parse: a variable, a
constant,
      // or an expression within parentheses. It includes syntax error checks for
      // unmatched parentheses or unexpected tokens.

      printf("Exit <factor> \n");
}
```

Code Output:

Test1:

```
maxi@maxis-MacBook-Air test % gcc test.c -o test1
test.c:16:15: warning: a function declaration without a prototype is deprecated in all versions of C and is treated a
s a zero-parameter prototype in C2x, conflicting with a previous declaration [-Wdeprecated-non-prototype]
FILE *in_fp, *fopen();
             ^
1 warning generated.
maxi@maxis-MacBook-Air test % ./test1
Next token is: 25, Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11, Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21, Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10, Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26, Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24, Next lexeme is /
Exit <factor>
Next token is: 11, Next lexeme is total
Enter <factor>
Next token is: -1, Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

Test2:

```
maxi@maxis-MacBook-Air test % gcc test.c -o test2
test.c:16:15: warning: a function declaration without a prototype is deprecated in all versions of C and is treated a
s a zero-parameter prototype in C2x, conflicting with a previous declaration [-Wdeprecated-non-prototype]
FILE *in_fp, *fopen();
             ^
1 warning generated.
maxi@maxis-MacBook-Air test % ./test2
Next token is: 10, Next lexeme is 50
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 10, Next lexeme is 47
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24, Next lexeme is /
Enter <expr>
Enter <term>
Enter <factor>
Error: Syntax token
Exit <factor>
Next token is: 11, Next lexeme is x
Enter <factor>
Next token is: -1, Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
maxi@maxis-MacBook-Air test %
```

Test3:

```
maxi@maxis-MacBook-Air test % gcc test.c -o test3
test.c:16:15: warning: a function declaration without a prototype is deprecated in all versions of C and is treated a
s a zero-parameter prototype in C2x, conflicting with a previous declaration [-Wdeprecated-non-prototype]
FILE *in_fp, *fopen();
             ^
1 warning generated.
maxi@maxis-MacBook-Air test % ./test3
Next token is: 25, Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11, Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21, Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10, Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 24, Next lexeme is /
Exit <factor>
Next token is: 11, Next lexeme is total
Enter <factor>
Next token is: -1, Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
Error: Missing closing parenthesis in file
Exit <factor>
Exit <term>
Exit <expr>
maxi@maxis-MacBook-Air test %
```